A

The Migration Prefetcher: Anticipating Data Promotion in Dynamic NUCA Caches

Javier Lira, Universitat Politècnica de Catalunya
Timothy M. Jones, University of Cambridge
Carlos Molina, Universitat Rovira i Virgili
Antonio González, Intel Barcelona Research Center, Intel Labs - UPC

The exponential increase in multicore processor (CMP) cache sizes accompanied by growing on-chip wire delays make it difficult to implement traditional caches with a single, uniform access latency. Non-Uniform Cache Architecture (NUCA) designs have been proposed to address this problem. A NUCA divides the whole cache memory into smaller banks and allows banks nearer a processor core to have lower access latencies than those further away, thus mitigating the effects of the cache's internal wires. Determining the best placement for data in the NUCA cache at any particular moment during program execution is crucial for exploiting the benefits that this architecture provides. Dynamic NUCA (D-NUCA) allows data to be mapped to multiple banks within the NUCA cache, and then uses data migration to adapt data placement to the program's behavior. Although the standard migration scheme is effective in moving data to its optimal position within the cache, half the hits still occur within non-optimal banks. This paper reduces this number by anticipating data migrations and moving data to the optimal banks in advance of being required. We introduce a prefetcher component to the NUCA cache that predicts the next memory request based on the past. We develop a realistic implementation of this prefetcher and, furthermore, experiment with a perfect prefetcher that always knows where the data resides, in order to evaluate the limits of this approach. We show that using our realistic data prefetching to anticipate data migrations in the NUCA cache can reduce the access latency by 15% on average and achieve performance improvements of up to 17%.

Categories and Subject Descriptors: B.3.2 [**Design Styles**]: Cache Memories; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Parallel Processors (CMPs)

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Memory hierarchy, cache memory, NUCA, migration, prefetching

ACM Reference Format:
Lira J., Jones T. M., Molina C., González A. 2011. The Migration Prefetcher: Anticipating Data Promotion in Dynamic NUCA Caches. ACM Trans. Architec. Code Optim. V, N, Article A (January YYYY), 21 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Researchers from both academia [Kim et al. 2002] and industry [Borkar et al. 2005] agree that future CMPs will accommodate large on-chip last-level caches. However, the exponential increase in multicore processor cache sizes accompanied by growing on-chip wire delays [Agarwal et al. 2000] make it difficult to implement traditional caches with a single, uniform access latency. Non-Uniform Cache Architecture (NUCA) designs [Kim et al. 2002] have been proposed to address this problem. A NUCA divides

This work is supported by the Spanish Ministry of Science and Innovation (MCI) and FEDER funds of the EU under the contracts TIN 2010-18368 and TIN 2007-68050-C03-03, the Generalitat de Catalunya under grant 2009SGR1250, EPSRC, the Royal Academy of Engineering and Intel Corporation.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

the whole cache memory into smaller banks and allows those that are located next to the cores to have lower access latencies than the banks that are further away, thus mitigating the effects of the cache's internal wires.

An important issue to be addressed for CMP NUCAs is *data placement*, which can be described as determining the most suitable location for data blocks in the NUCA space. This is not an easy problem to solve as different data blocks can experience entirely different degrees of sharing (i.e., the number of processors that access them) and exhibit a wide range of access patterns. Moreover, the access pattern of a given data block can also change during the course of program execution. Therefore a static placement strategy which fixes the location of each block for the entire application will perform sub-optimally. To address this issue, dynamic NUCA (D-NUCA) [Beckmann and Wood 2004], [Kim et al. 2002] allows data to be mapped to multiple banks within the NUCA cache, and then uses *data migration* to adapt its placement to the program behavior as it executes.

Prior research on D-NUCA has developed two main trends for dealing with data migration: 1) promotion and 2) optimal position. With promotion [Beckmann and Wood 2004], [Kim et al. 2002] the requested data is moved closer to the processor that initiated the memory request upon a hit in the NUCA cache. This scheme is especially effective in uniprocessor systems. However, in a CMP environment, multiple processors accessing shared data results in this data "ping-ponging" between NUCA banks.

Other migration schemes in the literature compute the optimal placement for a particular data block by taking into consideration dynamic parameters such as the usage of data or the processors that are currently sharing it [Hammoud et al. 2009], [Kandemir et al. 2008]. An alternative for D-NUCA are those NUCA designs based on S-NUCA configurations [Beckmann et al. 2006], [Hardavellas et al. 2009]. These designs avoid data migration by using external support, like OS or selective data replication, to solve the data placement issue. These schemes effectively reduce the ping-pong effect at the cost of significantly increasing complexity. In general, current migration schemes leverage locality of data to reduce the NUCA access latency for future accesses to the same data, but do not exploit data access patterns or data correlation. In this paper we apply these concepts to a traditional migration technique in order to anticipate and speed up data migration on D-NUCA architectures.

Existing data migration policies are effective in concentrating the most frequently accessed data in the NUCA banks with the shortest access latency (i.e. the optimal banks). However, a significant percentage of hits in the NUCA cache are still resolved in slower banks. In this paper, we complement the migration scheme with a simple prefetching technique in order to recognize access patterns to data blocks and anticipate data migration. We show that using data prefetching to anticipate data migrations in the NUCA cache can reduce the access latency by 15% on average and achieve performance improvements of up to 17%.

The remainder of this paper is structured as follows. Section 2 describes the baseline architecture assumed in this paper. In Section 3 the proposed mechanism is described in detail. Section 4 presents the experimental methodology we used, followed by the analysis of the results that is presented in Section 5. Related work is discussed in Section 6, and concluding remarks are given in Section 7.

2. BASELINE ARCHITECTURE

We assume an L2 cache with a Non-Uniform Cache Architecture (NUCA), derived from the Dynamic NUCA (D-NUCA) design by [Kim et al. 2002]. As in the original proposal we partition the address space across cache banks which are connected via a 2D mesh interconnection network. As illustrated in Figure 1, the NUCA storage is partitioned

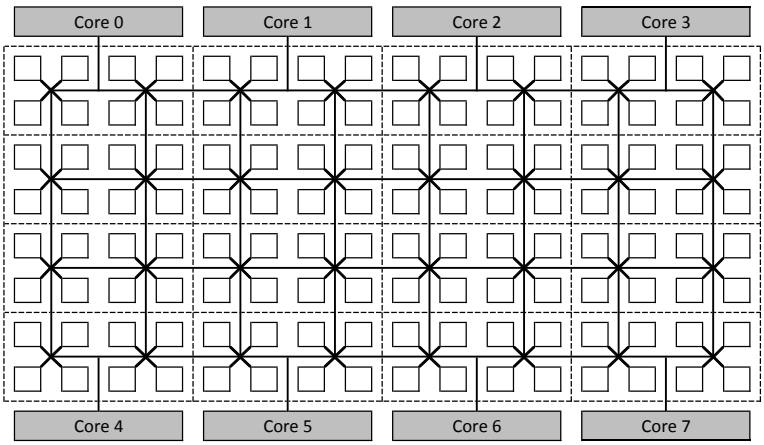


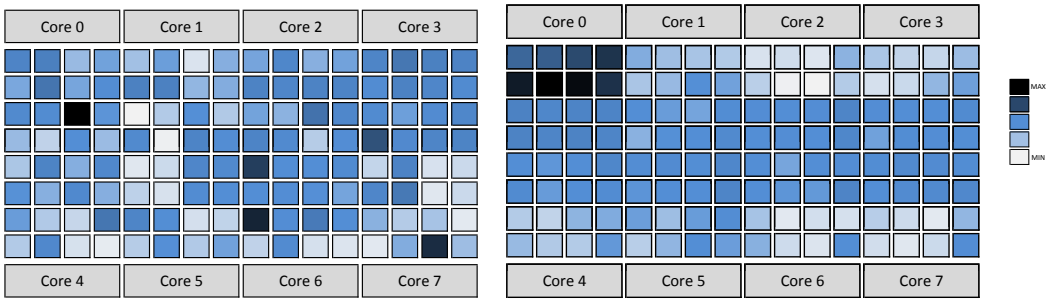
Fig. 1: Baseline architecture layout.

into 128 banks. D-NUCA allows migration of data towards the cores that use it the most. This distributes data blocks among the NUCA banks so data close to the cores is accessed the most often, thus reducing the access latency for future accesses to the same data.

Ideally, a data block would be mapped into any cache bank in order to maximize placement flexibility. However, the overheads of locating a data block in this scenario would be too large as each bank would have to be searched, either through a centralized tag store or by broadcasting the tag to all banks. To mitigate this, the NUCA cache is treated as a set-associative structure with each bank holding one "way" of the set. The banks which make up each set are called *banksets* and data blocks can be mapped to any bank within a single bankset. The NUCA banks that make up a bankset are organized into bankclusters within the cache (the dotted boxes in Figure 1). Each bankcluster consists of a single bank from each bankset. As shown in Figure 1, we assume a NUCA cache that is 16-way bankset associative, organized in 16 bankclusters. Of these, eight are located close to the cores (the local banks) and the other eight in the center of the NUCA cache (the central banks). Therefore, a data block has 16 possible placements in the NUCA cache (eight local banks and eight central banks).

An incoming data block from the off-chip memory is mapped to a specific bank within the cache. This is statically predetermined based on the lower bits from the data block's address. The LRU data block in this bank would be evicted if required. Once the data block is in the NUCA cache, the migration scheme determines its optimal position. As a migration policy, we assume *gradual promotion* that has been widely used in the literature [Beckmann and Wood 2004], [Kim et al. 2002]. This states that upon a hit in the cache the requested data block should move one-step closer to the core that initiated the memory request.

It is also necessary to determine how to find data within the NUCA cache. This is known as the *data search* scheme. The baseline D-NUCA design uses a two-phase multicast algorithm that is also known as *partitioned multicast*. First, it broadcasts a request to the *local bank* that is closest to the processor that launched the memory request, and to the eight *central banks*. If all nine initial requests miss, the request is sent, also in parallel, to the remaining seven banks from the requested data's bankset. Finally, if the request misses in all 16 banks, the request is forwarded to the off-chip memory.



(a) Static placement **(b)** D-NUCA with gradual migration

Fig. 2: Distribution of hits in the NUCA cache when core 0 sends memory requests.

Having described our baseline D-NUCA architecture, the next section presents our migration prefetcher in more detail.

3. THE MIGRATION PREFETCHER

This section describes our migration prefetcher in detail. We first motivate the need for this approach in Section 3.1. The implementation and the basic concepts of the prefetcher are described in Section 3.2. Then, we show several alternatives to deal with the prefetching strategies, accuracy and the data search scheme, which are described in Sections 3.3, 3.4 and 3.5, respectively. Finally, the actual prefetcher is presented in Section 3.6.

3.1. Motivation

To illustrate the need for our prefetching approach, consider Figure 2 which shows the distribution of hits within the NUCA cache for accesses made by core 0. Figure 2a shows a static placement of data which provides an even distribution of hits among all banks. Figure 2b, on the other hand, shows the same accesses when using a simple migration scheme, like *gradual promotion*. Note that the number of hits in both figures is exactly the same. Here, as expected, the NUCA banks that are closest to core 0 have the highest concentration of hits as data is migrated into them. However, despite this migration, half the hits from core 0 still find their data in the farther (i.e., non-optimal) banks.

This can partly be explained by data sharing. Some of the data required by core 0 is accessed by other cores too. Therefore, we would expect some of it to migrate towards those other cores, especially if they access the data more frequently than core 0. However, accessing these non-optimal banks from core 0 is costly in terms of both time and energy due to the search scheme employed to find the data. What we really require is a system to automatically bring this data closer to the core that will next need it, before it is actually required, instead of having to perform the search on the critical path when the memory request is actually initiated.

The aim of our prefetching technique is to increase the number of hits in the closest banks to each core by anticipating data migrations and bringing the data closer to the requesting core in advance of it being required. To do this we introduce a prefetcher into the NUCA cache that predicts the next memory request to hit in the NUCA cache based on the past. Figure 3 shows the potential for our prefetching strategy using our benchmark applications described in Section 4. Here, for each pair of consecutive memory accesses by a single core X, Y we show the fraction of accesses to the cache for

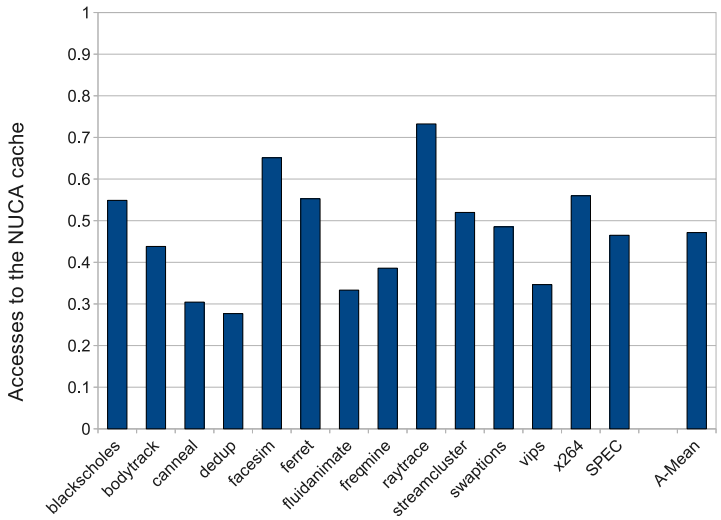


Fig. 3: Percentage of X → Y access patterns.

X where the next access by the same core is to the same address Y. In other words, the number of times an access to X is immediately followed by an access to the same Y. Figure 3 shows that, on average, almost 50% of the accesses exhibit this pattern and that this behavior is present across all applications, ranging from 30% to 70% of all accesses. We exploit this result in our prefetcher to predict the next memory address that will be required based on the current.

Note that the migration prefetcher is not a regular prefetcher in the sense that it does not bring data into the cache from main memory. Instead, this mechanism anticipates migrations by moving data blocks that are currently stored in the NUCA cache closer to the requesting core in advance of them being required. Moreover, unlike a regular prefetcher, this mechanism neither consumes memory bandwidth nor provokes replacements in the NUCA that could hurt performance. In general, a regular prefetcher and the migration prefetcher are orthogonal mechanisms that could be used simultaneously.

The potential benefits of a migration prefetcher are restricted to the difference between a near-bank and far-bank access latency. Traditional prefetchers, on the other hand, reduce the latency of an access to main memory, which is increasing with each generation. Therefore, a traditional prefetcher can afford gigantic structures to deal with complicated address correlation, whereas for our scheme this is not viable. In short, the migration prefetcher should use simple prefetching techniques, like the next access correlation we use in this paper.

3.2. How the prefetcher works

Figure 4 shows the NUCA cache including the prefetcher components. There are eight prefetchers (one per core) that are located in the cache controller, which is the entrance point to the NUCA from the L1 cache. A prefetcher consists of the *prefetcher slot (PS)*, which stores the prefetched data, a structure to manage the *outstanding prefetching request* and the *last request* sent from the L1, and the *Next Address Table (NAT)*, which keeps track of the data access patterns. For each address requested, the NAT stores

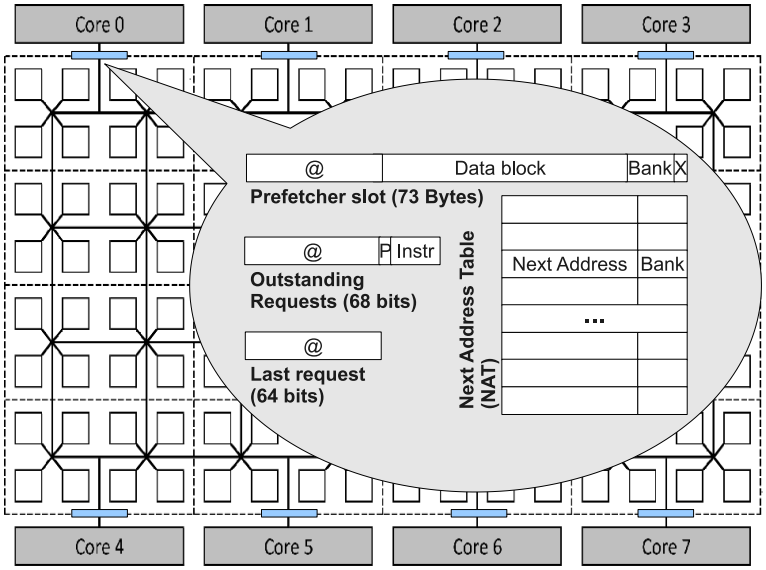


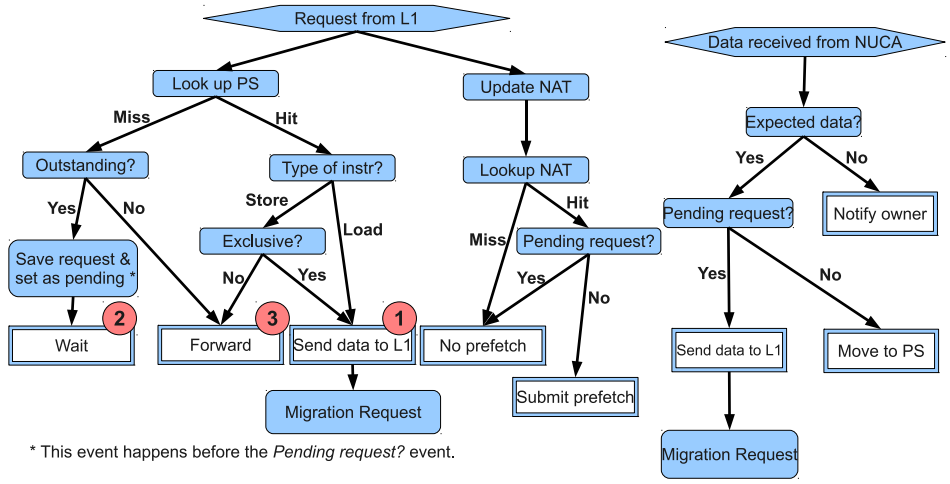
Fig. 4: Additional structures introduced to enable migration prefetching.

the next address to be accessed. In this section, for the sake of simplicity, we assume that the NAT is unlimited table which avoids address conflicts.

The prefetcher manages memory requests to the NUCA cache and keeps track of data access patterns. When a known pattern starts, the prefetcher predicts the next address and then prefetches it into the PS, and therefore closer to the requesting core. If the prediction was correct then the latency of the second access would be reduced since it would hit in this structure instead of a far-away bank.

Note that our prefetching technique is speculative. Therefore taking prefetching decisions into consideration when migrating data could lead to it being removed from its optimal position and cause unnecessary data movements. To address this situation, the PS always keeps a copy of the prefetched data on behalf of a bank in the NUCA. When there is a hit in the prefetcher (i.e., the PS holds the required data), it sends the data block to the L1 cache and then notifies the owner bank that it should migrate the data one step closer to the requesting core. In order to avoid interfering with the cache coherence protocol, data is copied to the prefetcher only if it is being shared by several L1 caches (all of them holding it in read-only mode), or the NUCA bank has the requested data in exclusive mode. Note that the prefetcher can only respond to stores if it gets the data block in exclusive mode, otherwise only loads can be satisfied. When the NUCA bank cannot send a copy of the requested data to the prefetcher, it notifies the prefetcher which records this information. Then, once the actual memory access is performed, although the prefetcher does not hold the requested data, it knows its bank position in the NUCA, speeding up access to the block.

When a request from the L1 cache arrives to the prefetcher (Figure 5a), it can be resolved in three ways: 1) If it hits in the PS, and the memory request is a load instruction or the PS has the data block in exclusive mode, the prefetcher sends the requested data to the L1 cache. This is the optimal situation because the memory request was serviced in the minimum amount of time due to the previous prefetch. 2) If the PS does not have the requested data, it still can hit on an outstanding prefetch



(a) Request from L1 cache arrives. **(b)** Prefetcher receives data.

Fig. 5: Flowcharts showing the prefetcher actions when receiving requests from the L1 cache and data from the NUCA cache.

request. Although the memory request will take longer than in the optimal case, there can be a reduction in the access latency in this situation. 3) If the request misses in both structures then it will be forwarded to the NUCA cache, just as if the prefetcher was not there. This situation also occurs if the prefetcher does have the data block but does not satisfy the coherence requirements for it, (e.g., the PS is storing the requested data in read-only mode and the core wants it in exclusive).

At the same time as checking the PS, the NAT is updated by storing the current address in the entry for the last request, then updating the last request field with the current address. If a pattern starting with the address of the current memory request exists in the NAT then the prefetcher submits a prefetch request and updates the outstanding request field. Note that if the prefetcher already has an outstanding prefetch request to the same memory location as the current address, it cannot submit a new prefetch request. Instead, the prefetcher must wait until the data comes back from the NUCA cache.

Submitting a prefetch request does not guarantee that the prefetcher gets any data. For example, the required data may not be in the NUCA cache. In this case, the prefetcher is notified, leading to case 3) above. Due to this we use only one slot to keep track of outstanding prefetch requests and this can be overwritten with a new prefetch request if the data is not found in the cache.

Having a single structure for managing outstanding prefetch requests is very convenient in our particular case as we model simple in-order processors in our baseline configuration. Bigger cores, however, may require dedicated hardware to deal with their specific features. For out-of-order processors, the effectiveness of the migration prefetcher would not significantly change. There are two main reasons for this. Firstly, our mechanism is implemented at the last level cache, so the memory requests that it receives are filtered by the L1 cache. Therefore the NUCA cache is only accessed for a small fraction of the total memory requests (i.e. those that miss in the L1) and these are likely to be similar for both in-order and out-of-order processors. Second, resolution of dependences in an out-of-order processor will be the same for iterations of the same piece of code. Other features in processors like SMT would

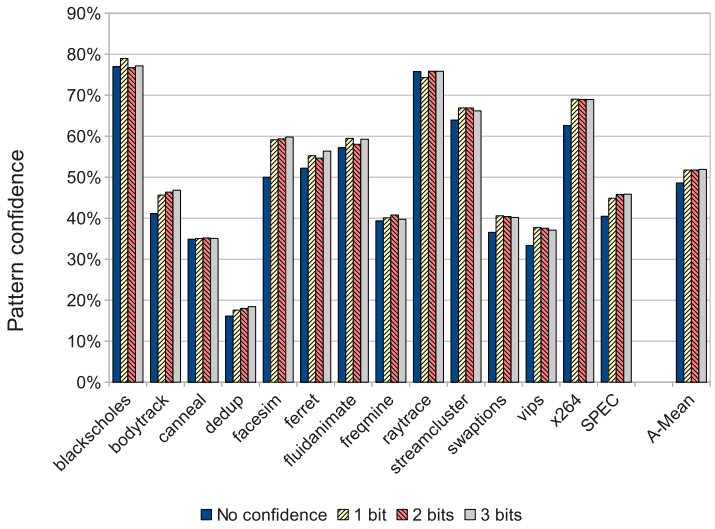


Fig. 6: Fraction of prefetch requests that were useful for varying confidence counter sizes.

require replicating some of the structures in the Migration Prefetcher to deal with the application behaviour.

Figure 5b illustrates the behavior of the prefetcher when it receives the prefetched data from the NUCA cache. If the prefetch request is still alive and there is a pending request, the prefetcher sends the received data to the L1 cache and it notifies the owner bank to migrate the prefetched data block one-step closer to the requesting core. Note that if the coherence requirements are not satisfied then the pending memory request must be forwarded to the NUCA cache. On the other hand, if there is no pending request then the received data is stored in the PS.

3.3. Prefetching strategies

Updating the NAT with the correct addresses is crucial for the prefetch requests to be useful in the future. In this section we propose including a saturating counter with each line in the NAT that shows the confidence of the corresponsing prediction. When updating the NAT, we first update and check the confidence counter. If the former address is the same as the new one the confidence counter is increased by one, otherwise it is decreased by one. If the modified confidence counter is greater than zero then the NAT will not be updated with the new address, instead keeping the old address. This strategy prevents the removal of a good prediction by a single miss. This is important, for example, when executing loops where there is a regular pattern of memory accesses. The loop will iterate many times but, on the final iteration, the prediction based on the last memory access will be incorrect. However, this will only occur once and we want to maintain the old prediction for the next time the loop is executed. Through the use of the confidence system we can ensure that the old prediction is not unnecessarily removed.

Figure 6 shows the fraction of prefetch requests that ended up being useful as the size of the confidence counter increases. This shows that implementing a confidence counter per NAT line effectively increases the rate of useful prefetch requests by 5%

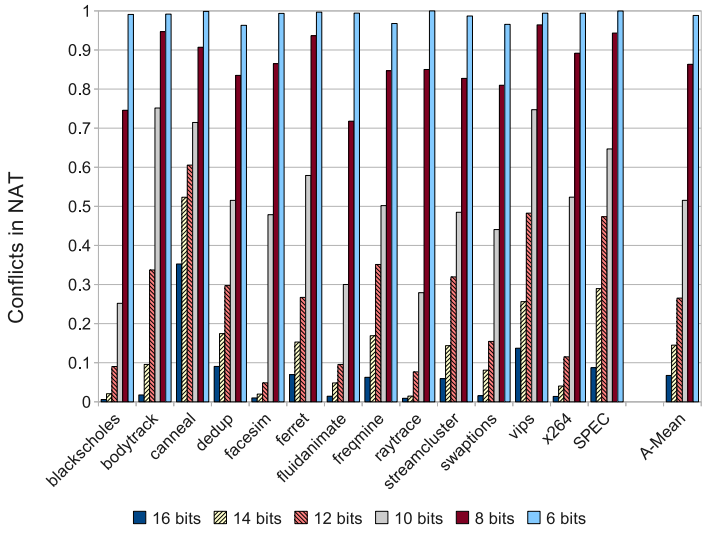


Fig. 7: Conflicts in the NAT table.

on average and up to 10% in the best case. However, we also observe that using more than one bit to implement the confidence system does not provide benefits, and can even reduce the performance of the system. This happens because some data blocks eventually change behavior and are followed by a different memory access. The greater the saturated counter, the longer the prefetcher takes to catch up with its current behavior.

Based on this analysis, we implemented a one-bit confidence system in the prefetcher. The prefetcher includes one extra bit per NAT line to represent the confidence counter and a comparator to determine whether the NAT line should be updated. We take these overheads into consideration when computing the prefetcher latency and energy consumption.

3.4. Accuracy

Up to this point we have considered an unlimited NAT. However, this is not feasible to implement, requiring 2^{58} lines of 58 bits for a 64-bit processor with data blocks of 64 bytes. Therefore this section analyses the accuracy loss that the prefetcher experiences if we assume an implementable NAT. Figure 7 shows the percentage of prefetch requests that are submitted using another address's information, caused by a conflict in the NAT. We show six configurations that use between 6 and 16 bits from the address to directly index into the NAT. This figure shows that conflicts on the NAT increase exponentially as the NAT's size decreases. For a very small NAT of 64 lines (6 bits), almost every prefetch uses the wrong information and the prefetcher is practically useless. However, using bigger tables (12-14 bits) means that only 25% of prefetch requests use erroneous information. The hardware overhead of using 12 bits to map addresses to the NAT is 232 KBytes, 29 KBytes per NAT. If we use 14 bits, the total overhead is almost 1 MByte.

A smarter implementation to reduce the hardware required to implement the NAT could take advantage of the locality found in applications, using the fact that the typical offset between two consecutive addresses is very small. Thus, we could consider

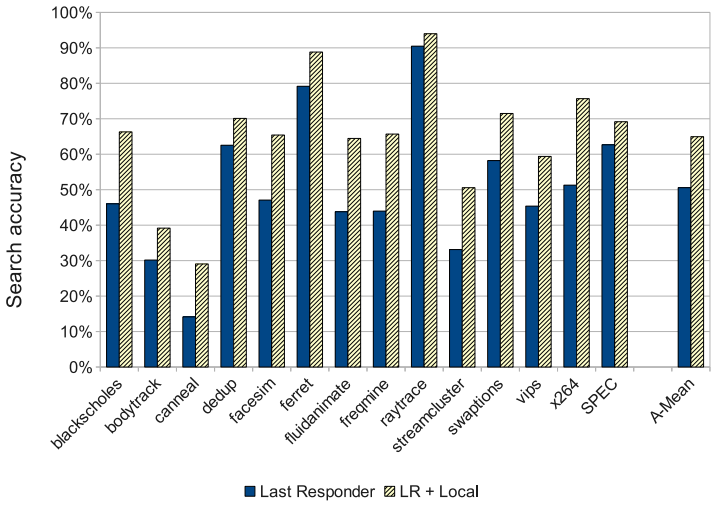


Fig. 8: Accuracy of the data search scheme.

storing the offset in the NAT instead the whole address. We believe that 16 bits would be enough to have good accuracy. Unfortunately, reducing the width of each NAT line is not as crucial as reducing its length. In this case, using 14 bits to map addresses to the NAT would make the NAT implementable. However, the accuracy loss caused by being unable to represent large offsets would prevent this scheme from achieving finer accuracy than the implementation proposed.

3.5. Lookup in NUCA

The dynamic placement of data in the NUCA cache makes the data search scheme the key challenge in D-NUCA architectures. As described in Section 2, the baseline architecture assumes the *partition multicast* algorithm to find data in the cache. This is a two-step access scheme that initially looks up the closest banks to the requesting core and, if the requested data is still not found, it looks up the rest. Unfortunately, using an access scheme like this to resolve prefetch requests could dramatically increase on-chip network traffic and, consequently, reduce the potential benefits of using a prefetcher in terms of performance and energy consumption.

In addition to predicting the next address, we also allow the prefetcher to predict the position in the NUCA cache where this data will be found. We experimentally observe that related data move together among bankclusters, therefore a significant percentage of consecutive memory requests are resolved by banks that belong to the same bankcluster. Based on this observation, for each line in the NAT we also keep the number of the bankcluster of the last responder that held the data belonging to the corresponding address. Therefore the prefetch request is sent only to one bank instead of to all 16 candidate banks. Figure 8 shows the accuracy of the search scheme compared to an oracle that always knows where the prefetched data is. This shows that using the last responder scheme means that, on average, half of the prefetch requests are found. However, this percentage is even higher (up to 90%) in some of the simulated applications, like *ferret* or *raytrace*. In order to enhance the accuracy of the access scheme without significantly increasing the network-on-chip traffic we consider also sending the prefetch request to the local bank when it is not the last responder.

In this case, as Figure 8 shows, the search accuracy increases to 70% compared to the oracle. We call this the *last responder and local* scheme.

3.6. Tuning the prefetcher

The prefetcher described in Section 3.2 introduces several challenges that must be met in order to provide a realistic implementation, such as the size of the NAT or the access scheme used when looking for data. We have analysed these challenges and propose a prefetcher with a *one-bit confidence system*, an *NAT table size of 29 KBytes* (12 addressable bits), and the *last responder and local* as a search scheme. The total hardware overhead of the actual prefetcher is 264 KBytes (33 KBytes per core), which represents less than 4% extra hardware compared to the baseline architecture. This includes all prefetcher structures, the confidence counter and the bankcluster identifier. We have modeled all prefetcher structures using CACTI and computed its global latency, indicating that it takes up to 2 cycles to perform the actions described in this section. Other aspects of the prefetcher, such as its influence in terms of performance and energy consumption, are described in Section 5.

4. EXPERIMENTAL METHODOLOGY

This section describes our simulator and benchmarks used to evaluate our migration prefetcher. We also explain the energy model used to calculate the energy consumed by our design.

4.1. Simulation Environment

We use the full-system execution-driven simulator, Simics [Magnusson et al. 2002], extended with the GEMS toolset [Martin et al. 2005]. GEMS provides a detailed memory-system timing model that enables us to model the NUCA cache architecture. Furthermore, it accurately models the network contention introduced by the simulated mechanisms. The simulated architecture is structured as a single CMP made up of eight in-order homogeneous cores. Each core is augmented with a split first-level cache (data and instruction). The second level of the memory hierarchy is the NUCA cache. In order to maintain coherency within the memory subsystem we used the MESIF coherence protocol, which is also used in the Intel® Nehalem processor [Molka et al. 2009].

Table I summarizes the configuration parameters used in our studies. The access latencies of the memory components are based on the models made with the CACTI 6.0 [Muralimanohar et al. 2007] modeling tool, this being the first version of CACTI that enables NUCA caches to be modeled.

Processors	8 - UltraSPARC IIIi
Frequency	1.5 GHz
Integration Technology	45 nm
Block size	64 bytes
L1 Cache (Instr./Data)	32 KBytes, 2-way
L2 Cache (NUCA)	8 MBytes, 128 Banks
NUCA Bank	64 KBytes, 8-way
L1 Latency	3 cycles
NUCA Bank Latency	4 cycles
Router Latency	1 cycle
Avg Offchip Latency	250 cycles

Table I: Configuration parameters.

In order to evaluate this mechanism, we assume two different scenarios: 1) Multi-programmed and 2) Parallel applications. The former executes a set of eight different SPEC CPU2006 [SPEC 2006] workloads with the *reference* input in parallel. Table II outlines the workloads that make up this scenario. The latter simulates the whole set of applications from the PARSEC v2.0 benchmark suite [Bienia et al. 2008] with the *simlarge* input data sets. This suite contains of 13 programs from many different application areas such as image processing, financial analytics, video encoding, computer vision and animation physics, among others.

astar	gcc	lbm	mcf
milc	omnetpp	perlbench	soplex
Reference input			

Table II: Multi-programmed scenario.

The methodology we used for simulation involved first skipping both the initialization and thread creation phases, and then fast-forwarding while warming all caches for 500 million cycles. Finally, we performed a detailed simulation for 500 million cycles. As a performance metric, we used the aggregate number of user instructions committed per cycle, which is proportional to the overall system throughput [Wenisch et al. 2006].

4.2. Energy Model

In this paper we evaluate the energy consumed by the NUCA cache and the off-chip memory. To do so, we used a similar energy model to that adopted by [Bardine et al. 2007]. This allowed us to also consider the total energy dissipated by the NUCA cache and the additional energy required to access the off-chip memory. The energy consumed by the memory system is computed as follows:

$$E_{total} = E_{static} + E_{dynamic}$$
$$E_{sta} = E_{S\_noc} + E_{S\_banks} + E_{S\_prefetcher}$$
$$E_{dyn} = E_{D\_noc} + E_{D\_banks} + E_{D\_prefetcher} + E_{off-chip}$$

We used models provided by CACTI 6.0 [Muralimanohar et al. 2007] to evaluate static energy consumed by the memory structures (E_{S_banks} and $E_{S\_prefetcher}$). CACTI has been used to evaluate dynamic energy consumption as well, but GEMS [Martin et al. 2005] support is required in this case to ascertain the dynamic behavior in the applications (E_{D_banks} and $E_{D\_prefetcher}$). GEMS also contains an integrated power model based on Orion [Wang et al. 2002] that we used to evaluate the static and dynamic power consumed by the on-chip network (E_{S_noc} and E_{D_noc}). Note that the extra messages introduced by the prefetcher into the on-chip network have also been accurately modeled by the simulator. The energy dissipated by the off-chip memory ($E_{off-chip}$) was determined using the *Micron System Power Calculator* [Micron 2009] assuming a modern DDR3 system (4GB, 8DQs, Vdd:1.5v, 333 MHz). Our evaluation of the off-chip memory focused on the energy dissipated during active cycles and isolated this from the background energy. This study shows that the average energy of each access is 550 pJ.

As an energy metric we used the energy consumed per memory access. This is based on the energy per instruction (EPI) [Grochowski et al. 2004] metric which is commonly used for analysing the energy consumed by the whole processor. This metric works independently of the amount of time required to process an instruction and is ideal for throughput performance.

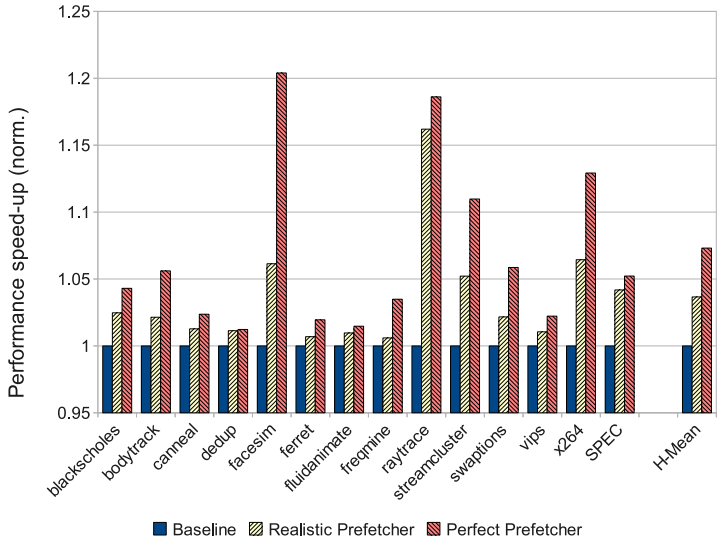


Fig. 9: Performance results.

5. RESULTS AND ANALYSIS

This section analyses the impact on performance and energy consumption of using the migration prefetcher on a D-NUCA architecture. We evaluate two versions of our prefetcher on a D-NUCA cache that uses *gradual promotion* as the migration policy and compare them to the baseline configuration, which is described in Section 2. These are a realistic implementation of the migration prefetcher and a perfect prefetcher. The first implements the *one-bit confidence system*, its *NAT table size is 29 KBytes* (12 addressable bits), and it uses *last responder and local* as a search scheme. On the other hand, the perfect prefetcher also implements the *one-bit confidence system*, but has an unlimited NAT table and uses a perfect search scheme. This represents an oracle that knows exactly where any data block resides within the NUCA cache at any time. Although unrealistic and unimplementable in practice, the perfect approach shows the potential benefits of prefetching for data migration.

5.1. Performance Analysis

Figure 9 shows the performance improvement obtained with the migration prefetcher. On average, we observe that the realistic implementation outperforms the baseline configuration by 4%, while the perfect prefetcher obtains a 7% performance improvement. This figure shows that, in general, the prefetcher is often able to find data access patterns in the simulated applications and exploit them for performance gains. Furthermore, it achieves speed-ups in all benchmarks and the multi-programmed mix (SPEC in Figure 9). In the parallel applications particularly, the realistic migration prefetcher achieves performance improvements of over 5% in four benchmarks (*facesim*, *streamcluster*, *x264*, and 17% in *raytrace*). On the other hand, when considering the multi-programmed environment, both versions of the migration prefetcher, perfect and realistic, outperform the baseline configuration by 5%. In many cases the realistic prefetcher achieves a performance improvement that approaches the perfect prefetcher (e.g., *raytrace* and *dedup*). In others, such as *facesim*,

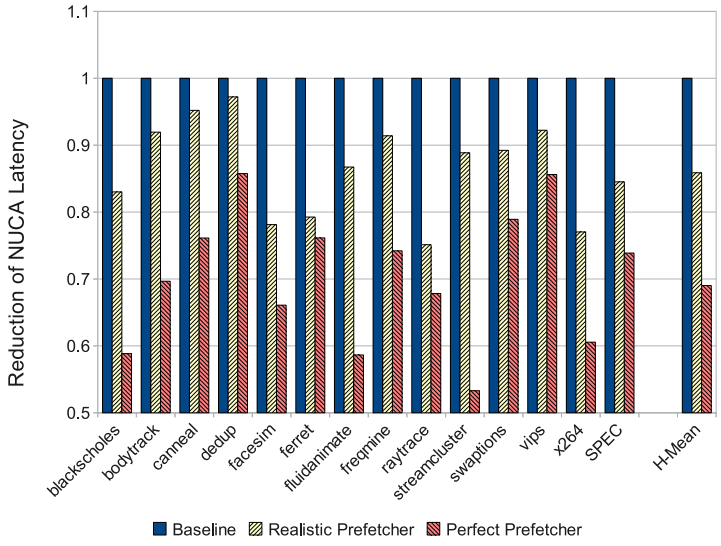


Fig. 10: Reduction of NUCA access latency.

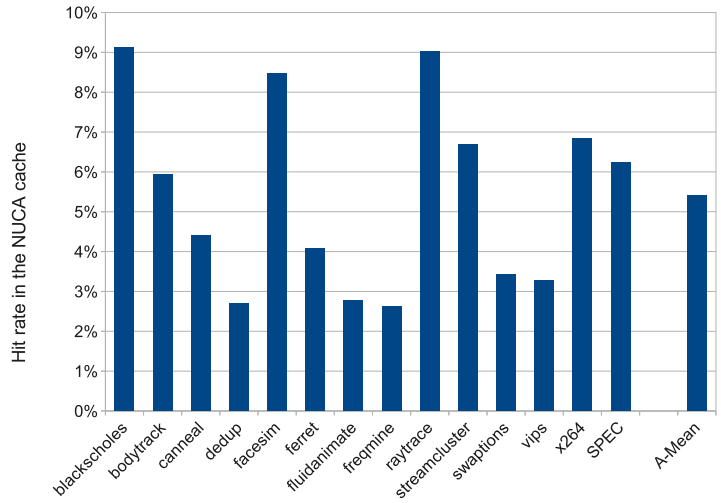


Fig. 11: Hit rate in the NUCA cache.

the improvement is less impressive when using the realistic scheme. We analyse the reasons for this later in this section.

The key reason for the performance improvements is a reduction in the access latency to the NUCA cache on a hit in the prefetcher. Figure 10 shows that the perfect prefetcher reduces the NUCA access latency by 30%, on average, while the reduction with the realistic implemetation is 15%. These results show the ability of the prefetcher to recognize data access patterns and anticipate data migrations in order to

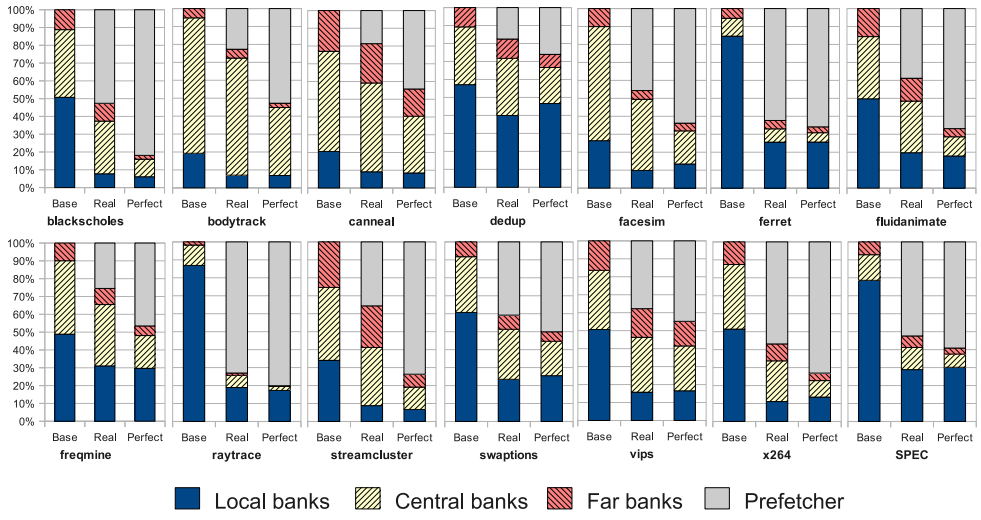


Fig. 12: Distribution of hits in the NUCA cache.

increase the number of memory requests satisfied with the optimal latency. Note that the realistic prefetcher succeeds in taking advantage of high temporal locality found in most of applications. This allows our implementation to deal with large amount of data correlation, even using small NAT structures. Unfortunately, the performance benefits of using this mechanism are restricted by the overall hit rate in the NUCA cache, which is small in some PARSEC applications (see Figure 11). For example, the access latency for *fluidanimate* is reduced by over 40% by using the perfect prefetcher. However, this only translates into a 2% performance increase for this application due to its low hit rate of 3%.

Figure 12 illustrates how hits in the NUCA cache are distributed among its different sections. For the sake of clarity we split the NUCA banks into three types which correspond to their distance from the requesting core. These are the *local banks* which are the 8 banks closest to the core, the *central banks* which are the 8 clusters (64 banks) in the center of the NUCA, and the *far banks* that are all other banks. Note that the migration prefetcher can only allocate one data block per core. However, as Figure 12 shows, it effectively services the vast majority of hits, even with the realistic implementation. This clearly explains why this mechanism outperforms the baseline configuration, overcoming the overheads of continuously submitting prefetch requests.

In addition, Figure 12 shows that the realistic implementation deals with as many data access patterns from local banks as the perfect prefetcher. This is because the access scheme used by the realistic prefetcher always sends the prefetch request to these banks first. In order to deal with patterns in the other banks, the realistic prefetcher sends the request to the last responder. This figure shows that it successfully prefetches and migrates some of the patterns from the central or far banks to the prefetcher. Compared to the perfect prefetcher, however, we find that there are a significant number of patterns that the realistic implementation cannot deal with.

Figure 13 shows the sources of prefetch data. It is clear from this that the perfect prefetcher obtains more data from the central and far-away banks than the realistic implementation. This is because the realistic prefetcher only accesses the local bank and last responder to find the required data. Using this simple search scheme, the

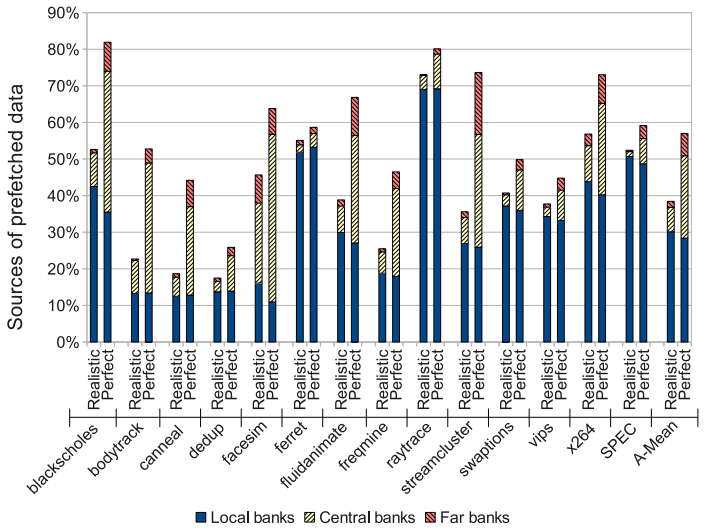


Fig. 13: Sources of prefetched data.

realistic approach gets about 20% of prefetched data from the non-optimal banks. On the other hand, the perfect prefetcher, that uses an oracle to know where in the NUCA cache is the prefetched data, shows the potential of our mechanism to deal with data located in non-optimal banks. It outlines that the realistic prefetcher could get up to 50% of prefetched data from central and far-away banks. Based on this observation, we conclude that far from the size of the NAT table being the limitation to this approach, the main challenge that needs to be addressed to reduce the difference between the perfect and realistic implementations is the data access scheme and its effectiveness in finding data access patterns in non-local banks. An example of this can be found by analysing how the two versions of the migration prefetcher behave in front of *facesim* and *raytrace*. These are the two applications that achieve the highest performance improvement when using the perfect prefetcher. In Figure 12, on one hand, we observe that *raytrace* fulfills almost 90% of its hits in the local banks. Therefore both prefetcher versions can get the data access patterns from there and, consequently, the realistic prefetcher achieves almost the same performance improvement as the oracle. Considering *facesim*, on the other hand, most of the hits in the NUCA cache happen in the central banks. The realistic prefetcher in this case is not able to obtain all data access patterns and ends up improving performance by 5% compared to the baseline configuration, whereas the perfect prefetcher achieves 20% performance improvement.

5.2. Energy Consumption Analysis

Here, we analyse the impact of the overheads that the migration prefetcher introduces to the baseline D-NUCA configuration. Figure 14 shows the on-chip network contention of the simulated mechanism by discriminating the traffic introduced by the prefetcher. The figure shows that, on average, neither of the two versions of the migration prefetcher increase the on-chip network contention. Although we would expect the prefetcher to significantly increase the on-chip network traffic due to the large numbers of prefetch requests that this mechanism sends, in actual fact every hit in the prefetcher saves up to 16 memory requests to the NUCA banks. To take advantage of this and therefore avoid increasing the on-chip network traffic, the

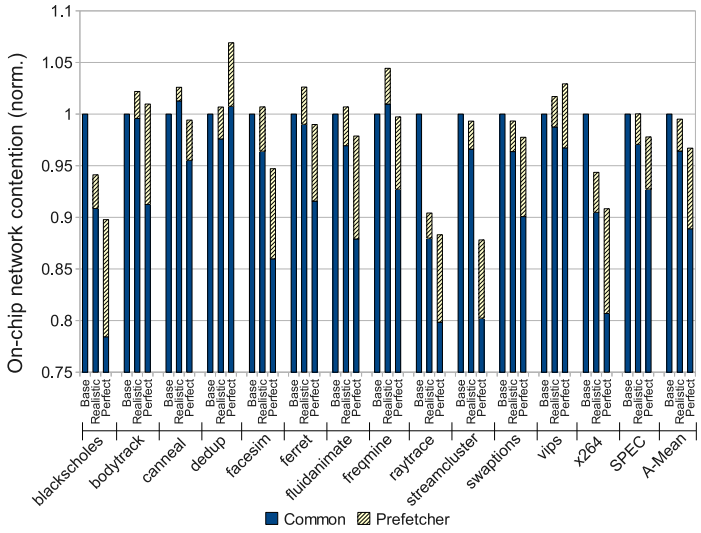


Fig. 14: Quantification of the on-chip network traffic.

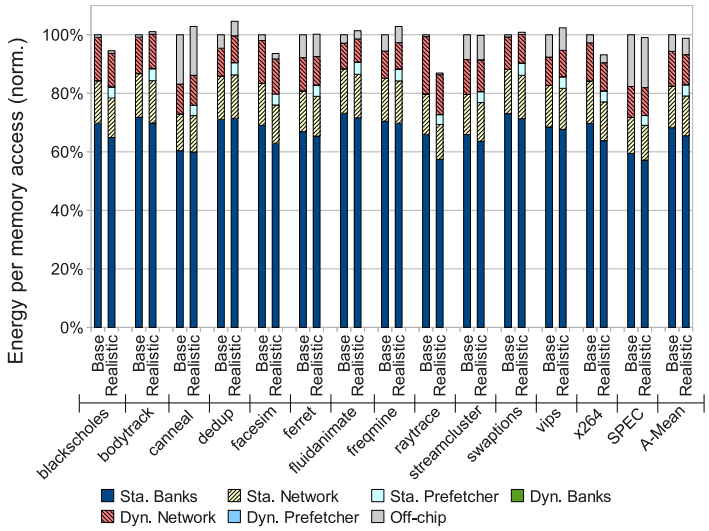


Fig. 15: Energy consumed per memory access.

migration prefetcher must be effective. Figure 14 also illustrates that the realistic implementation of the prefetcher introduces fewer extra messages to the network. This is because the perfect prefetcher always finds the prefetched data, and thus receives the corresponding data block. On the other hand, if the realistic prefetcher could not find the required data it must forward the memory request to the NUCA cache. Therefore the common on-chip network traffic increases compared to the perfect implementation.

Considering the energy consumption, Figure 15 illustrates that the migration prefetcher consumes about the same amount per memory access as the baseline configuration. For the sake of simplicity we remove the perfect prefetcher from this analysis because we cannot model an infinite NAT. Static energy is the major contributor to the overall energy consumed by caches, and this case is not an exception. In general we find that although the migration prefetcher introduces 264 KBytes to implement the required structures, it achieves a significant enough reduction in dynamic energy consumption in the on-chip network to counterbalance these overheads. As happens with the on-chip traffic, the amount of energy consumed by the prefetcher depends on its effectiveness. We can clearly see that the migration prefetcher achieves the highest reduction in energy consumption per memory access in applications where it also obtains the highest performance benefits, like *facesim*, *raytrace* or *x264*.

5.3. Summary

This section has analysed the performance and energy consumption of our migration prefetcher, showing that the realistic implementation reduces the average NUCA latency by 15%. This translates into an overall speed-up of 4% across all benchmarks, up to 17% in the case of *raytrace*. Furthermore, an analysis of the banks that are accessed in each scheme shows it is able to service significant numbers of hits from within the prefetcher itself, providing further evidence of the benefits that this technique can bring.

6. RELATED WORK

Prefetching is a well known technique for boosting program performance by moving data to a nearer cache level to the core before it is required by the execution units of the processor [Byna et al. 2009], [VanderWiel and Lilja 1996]. The most relevant approaches to our work are *Tagged Prefetching* [Smith 1982] that tries to capture accesses to consecutive memory lines, *Stride Prefetching* [Chen and Baer 1995] that tries to capture accesses at the same distance from one to next and *Correlation Prefetching* [Nesbit and Smith 2004] that tries to identify groups of memory accesses that are related under a common pattern against time.

Using address correlation for prefetching data from main memory has been studied in the literature for decades [Charney and Reeves 1995], [Joseph and Grunwald 1997], [Hu et al. 2003]. These works usually require large structures to deal with fine-grained address correlation, and therefore achieve high accuracy with the implemented prefetcher. The migration prefetcher is different to traditional prefetchers in the sense that our scheme moves data that is already in the NUCA from one bank to another, closer to the requesting core, while traditional prefetchers bring data from the main memory. The migration prefetcher enables a close-bank hit latency for data that is actually in distant banks. Its benefits are therefore restricted to the difference between a near-bank and far-bank access latency. Traditional prefetchers, on the other hand, reduce the latency of an access to main memory, which is increasing with each generation. Therefore, a traditional prefetcher can afford gigantic structures to deal with complicated address correlation, whereas for our scheme this is not viable. In short, the migration prefetcher should use simple prefetching techniques, like the next access correlation we use in this paper.

Prefetching in a NUCA was first analysed by [Beckmann and Wood 2004]. They evaluated stride-based prefetching between the L2 cache and memory and between L1 and L2 caches. In this paper, we apply prefetching in the NUCA cache to move data that is likely to be accessed close to the requesting core, but at the NUCA level. This mechanism, therefore, does not provoke replacements with prefetched data.

Migration techniques in D-NUCA have been widely analysed in the literature. [Kim et al. 2002] introduced the D-NUCA concept on uniprocessors and proposed a simple migration mechanism known as *gradual promotion*. This approach begins with a hit in the NUCA cache, and moves the requested data one-step closer to the processor that initiated the memory request. [Beckmann and Wood 2004] demonstrated that block migration is less effective for CMPs because 40-60% of hits in commercial workloads are satisfied in the central banks. In this paper, we show how prefetching can help migration to reduce the hit rate in the non-optimal banks.

[Eisley et al. 2008] proposed a scalable migration technique that tries to allocate data in free/invalid cache slots instead of sending data off-chip. [Kandemir et al. 2008] took a different point of view. They observed that 50% of data fetched from of-chip memory to a shared L2 NUCA cache was used more heavily by a different core from the one that originally requested the data. Based on this observation, they focused on minimizing the number of migrations with a placement mechanism that tries to place data in an optimal position. In the same vein, [Hammoud et al. 2009] predicted the optimal location of data by monitoring the behavior of programs. Finally, [Chaudhuri 2009] proposed a coarse-grained data migration mechanism assisted by the operating system that monitors access patterns to decide where and when an entire page of data should be migrated.

A common characteristic of these migration techniques is that they move data to the optimal position with the cache to reduce the access latency for future accesses to the same data. In our work, however, we analyse related data to anticipate migrations, and thus reduce access latency for future accesses not only to the same data, but also to its followers. Moreover, although in this paper we analysed this technique with *gradual migration*, we could implement it in conjunction with any other migration technique proposed in the literature.

7. CONCLUSIONS

Existing data migration policies for NUCA caches succeed in concentrating the most frequently accessed data in the NUCA banks with the smallest access latency. However, there is still a significant percentage of hits in the NUCA cache that are resolved in slower banks.

In order to address this situation, we have proposed a *migration prefetcher*. This recognizes data access patterns to the NUCA cache, and then anticipates data migrations, prefetching a copy of data and holding it close to the core. Using the migration prefetcher, therefore, the number of hits in the NUCA cache that take the optimal access latency increases. We find that a perfect prefetcher reduces the NUCA access latency, on average, by 30%, and improves performance by 7% compared to the baseline configuration. Assuming a more realistic approach, the migration prefetcher still achieves an overall latency reduction of 15%, and outperforms the baseline configuration by 4%, or up to 17% in one application.

REFERENCES

AGARWAL, V., HRISHIKESH, M. S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate vs. ipc: The end of the road for conventional microprocessors. In *Procs. of the 27th International Symposium on Computer Architecture*.

BARDINE, A., FOGLIA, P., GABRIELLI, G., AND PRETE, C. A. 2007. Analysis of static and dynamic energy consumption in nuca caches: Initial results. In *Procs. of the Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*.

BECKMANN, B. M., MARTY, M. R., AND WOOD, D. A. 2006. Asr: Adaptive selective replication for cmp caches. In *Procs. of the 39th Annual IEEE/ACM International Symposium of Microarchitecture*.

BECKMANN, B. M. AND WOOD, D. A. 2004. Managing wire delay in large chip-multiprocessor caches. In *Procs. of the 37th International Symposium on Microarchitecture*.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The parsec benchmark suite: Characterization and architectural implications. In *Procs. of the International Conference on Parallel Architectures and Compilation Techniques*.

BORKAR, S. Y., DUBEY, P., KAHN, K. C., KUCK, D. J., MULDER, H., PAWLOWSKI, S. S., AND RATTNER, J. R. 2005. Platform 2015: Intel processor and platform evolution for the next decade. *Technology@Intel Magazine*.

BYNA, S., CHEN, Y., AND SUN, X. H. May 2009. Taxonomy of data prefetching for multicore processors. *Journal of Computer Science and Technology 24(3)*, Pages 405–417.

CHARNEY, M. J. AND REEVES, A. P. Feb 1995. Generalized correlation based hardware prefetching. Tech. Rep. EE-CEG-95-1, Cornell University.

CHAUDHURI, M. 2009. Pagenuca: Selected policies for page-grain locality management in large shared chip-multiprocessors. In *Procs. of the 15th International Symposium on High-Performance Computer Architecture*.

CHEN, T. F. AND BAER, J. L. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers 44,* 5.

EISLEY, N., PEH, L. S., AND SHANG, L. 2008. Leveraging on-chip networks for cache migration in chip multiprocessors. In *Procs. of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

GROCHOWSKI, E., RONEN, R., SHEN, J., AND WANG, H. 2004. Best of both latency and throughput. In *Procs. of the 22nd Intl. Conference on Computer Design*.

HAMMOUD, M., CHO, S., AND MELHEM, R. 2009. Acm: An efficient approach for managing shared caches in chip multiprocessors. In *Procs. of the 4th Intl. Conference on High Performance and Embedded Architectures*.

HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. 2009. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Procs. of the 36th International Symposium on Computer Architecture*.

HU, Z., MARTONOSI, M., AND KAXIRAS, S. 2003. Tcp: Tag correlating prefetchers. In *Procs. of the 9th Intl. Symp. on High-Performance Computer Architecture*.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using markov predictors. In *Procs. of the 24th International Symposium on Computer Architecture*.

KANDEMIR, M., LI, F., IRWIN, M. J., AND SON, S. W. 2008. A novel migration-based nuca design for chip multiprocessors. In *Procs. of the International Conference on Supercomputing*.

KIM, C., BURGER, D., AND KECKLER, S. W. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Procs. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. *Simics: A Full System Simulator Platform*. Vol. 35-2. Computer, 50–58.

MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. In *Computer Architecture News*.

MICRON. 2009. System power calculator. In $http://www.micron.com/$.

MOLKA, D., HACKENBERG, D., SCHONE, R., AND MULLER, M. S. 2009. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Procs. of the International Conference on Parallel Architectures and Compilation Techniques*.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. P. 2007. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Procs. of the 40th International Symposium on Microarchitecture*.

NESBIT, K. J. AND SMITH, J. E. 2004. Data cache prefetching using a global history buffer. In *Procs. of the 10th Intl. Symp. on High-Performance Computer Architecture*.

SMITH, A. J. 1982. Cache memories. *ACM Computing Surveys 14,* 3.

SPEC. 2006. Spec cpu2006. In $http://www.spec.org/cpu2006$.

VANDERWIEL, S. AND LILJA, D. J. 1996. A survey of data prefetching techniques. In *Procs. of the 23rd International Symposium on Computer Architecture*.

WANG, H. S., ZHU, X., PEH, L. S., AND MALIK, S. 2002. Orion: A power-performance simulator for interconnection networks. In *Procs. of the 35th International Symposium on Microarchitecture*.

WENISCH, T. F., WUNDERLICH, R. E., FERDMAN, M., AILAMAKI, A., FALSAFI, B., AND HOE, J. C. 2006. Simflex: Statistical sampling of computer system simulation. *IEEE Micro 26,* 4, 18–31.